STATIC CONTEXTURAL ANALYSIS OF PASCAL PROGRAMS WITH L-ATTRIBUTE GRAMMARS

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By

KARNAL SINGH

to the
DEPARTMENT OF COMPUTER SCIENCE
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

AUGUST, 1981

CERTIFICATE

This is to certify that the thesis entitled "STATIC CONTEXTUAL ANALYSIS OF PASCAL PROGRAMS WITH L-ATTRIBUTE GRAMMARS" has been carried out by Sri Karnal Singh under my supervision and has not been submitted elsewhere for the award of a degree.

Kanpur
August 1981

Kesav V. Nori
Assistant Professor
Computer Science Programme
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

CS- 1981- M- SIN- STA

ACKNOWLEDGEMENTS

Words fail me when I try to express my profound sense of gratitude for my guide Prof. K.V. NORI. His ready anticipation of my difficulties, both academic and non-academic and his irreppressible zeal for solving them were instrumental in the successful completion of my project.

I can not do justice if I try to formally thank Pukhraj Kachhwaha, in whom I found a dream of a friend and colleague rolled into one. His thesis being a complement to mine, we have spent many a night and day working together in perfect synchronism.

Then there were my classmates, especially Santanu Datta and Deepak Sherlekar, with whom I spent many a fruitful discussion and whose camaraderie saw me through thick and thin.

My wingmates (H-middle) would certainly deserve a special mention.

I would also like to thank all my other friends, whose association has made my stay here, a memorable one.

Finally I thank Mr. G.L. Misra for the excellent typing.

Kanpur
AUGUST 1981

- KARNAL SINGH

NAME OF THE CANDIDATE : KARNAL SINGH
DEPARTMENT : Computer Science
PROGRAMME : M. Tech.
ROLL NUMBER : 911106
THESIS SUPERVISOR : Mr. K.V. NORI
TITLE OF THE THESIS : STATIC CONTEXTUAL ANALYSIS OF PASCAL PROGRAMS WITH L-ATTRIBUTE GRAMMARS.

ABSTRACT

This thesis deals with the numerous problems encountered in contextual analysis of the ACTION part of PASCAL programs, on the basis of the context synthesized during the declaration part. The principal issues tackled are type compatibility, label processing and aliasing. Special cases concerning modification of controlling elements within control structures, type compatibility of recursively defined data structures and illegal uses are also solved.

The machanism used for solving the above problems is an extension of L-Extended Attribute grammars. This provides a formal basis for our solution, leading to a systematic, modularised development which is amenable to a formal proving technique. The end result is a working context sensitive analyser meeting the specifications defined in the language standard.

CONTENTS

CHAPTER I

PROLOGUE

1.1 MOTIVATION

We are interested in the systematic development of processors that perform context sensitive analysis (CSA) for programming languages. To this end, we must show that it correctly accepts or rejects an input string, based on the specification of the semantics of the language.

The strong formal background of Context-Free Grammars (cfg) for exact syntactic specification of programming languages makes automatic generation of parsers feasible task. However, inspite of the presence of various mechanisms for formal specification of context sensitive aspects of a programming language, none of them is strong enough to produce an automatic context sensitive analyser generator system - this is so because the transformation rules for conversion from such a formal specification to programs are still not well established. We obtain a formal specification for the different context sensitive issues of a programming language in a manner suitable for ready implementation. The formal specification facilitates the proof of correctness of the developed program since the validity of our basic transformation rules ensure the correctness of the entire program. The formal specification even reveals the hidden semantic irregularities of a programming language and thus helps in correct implementation of the defined language.

1.2 Introduction :

We choose the programming language STANDARD PASCAL, ISO defined [SIG80] for the systematic development of CSA phase of its compiler.

This thesis deals with numerous problems encountered in contextual analysis arising in the ACTION PART of PASCAL programs based on the context already synthesized during declaration part (PUK 81) of the PASCAL program. Corresponding to the use of an object in the ACTION PART of a PASCAL program, the contextual analysis is performed on the attributes of the object defined in the context synthesized during DECLARATION PART and the validity of the use of the object, as per its definition, is determined.

The main problems encountered in CSA are concerning the issues of type compatibility, label processing and ALIASING. We also discuss certain special cases concerning modification of controlling elements within control structures, type compatibility of recursive data structures, illegal uses etc.

The CSA phase of our compiler uses the formal specification mechanism of L-EAG's. We have chosen this mechanism for formal specification as it is best suitable out of a few available. The work presented in this thesis is a completion of the task undertaken in (PUK 81).

1.3 Structure of the thesis.

Chapter 2 discusses various issues concerning extension of context and the problem of compatibility. Chapter 3 deals with label processing-problem definition and its solution. In Chapter 4 we have described the problem of ALIASING - its causes and detection. We discuss a few special cases of CSA in Chapter 5. Finally Chapter 6 describes possible extension to our scheme and scope of further work and improvements.

CHAPTER II

EXTENSION OF CONTEXT AND THE PROBLEM OF COMPATIBILITY

2.1 Processing with statements to extend the local context :

2.1.1. Structure of the problem :

(a) A statement of form :

with A1, A2, An do STATEMENT

must be treated as :

```
with A1 do
    with A2 do
        :
        with An do STATEMENT
```

(b) The identifiers belonging the local environment of a record encountered in a WITH statement must be accessible to the WITH statement and they override the visibility of context built up till now.

2.1.2 Solution :

Each occurrence of a record in a RECORD VAR-LIST of the WITH statement is considered as the start of a new block whose local environment is same as the local environment of the record. The life time of this temporary extension of context is till the end of the WITH statement.

It is implemented by putting the local environment of the RECORD VARIABLE on the top of the stack (SYMBOL TABLE) and keeping the name of the RECORD VARIABLE in the SYMBLKINFO.

2.2 Type Compatibility and operator Identification Problem.

2.2.1 Structure of problem :

The problem of type compatibility arises during the processing of procedure (function) invocations. It also occurs, with possible variations in the processing of operands in expressions and in assignment statements. The latter two issues give rise to the related problem of operator Identification wherein we discriminate between the various meanings of overloaded operator symbols.

2.2.1.1 Type compatibility for user defined procedures (functions) is defined as :

(1) Var parameter :

(a) Actual parameters shall be VAR-ACCESS (not an expression).

(b) The components of variables of any type designated packed shall not be used as actual variable parameters.

(c) The formal and actual parameters shall be of same type.

(d) If the formal parameter type is a CONFORMANT ARRAY SCHEMA (CAS), then an actual parameter should be conformable with CAS. [SIG 80].

(2) Value parameter :

(a) Actual parameter shall be an expression whose value is assignment compatible [SIG 80] with the corresponding formal parameter.

(3) Procedural parameter and functional parameter [SIG 80].

2.2.1.2 Type compatibility issues for user defined procedures (functions) has to be augmented to accomodate the special features of the parameters of system defined procedures (functions) and operators. These features are as follows :

(a) Class of Parameters :

(a.1) Polymorphic : Formal parameter is defined to have set of types and type of actual parameter must be compatible with a type in the set.

Example : Read (a)
where type of a can be integer, real, char or string.

(a.2) Polyadic : Zero or more instances of an actual parameter is allowed for one formal parameter.

Example : Readln; Readln (a,b,c);

(a.3) Default : There may or may not exist an actual parameter corresponding to the default parameter. If there is no actual parameter then the default value of parameter is chosen.

```
Example :  Type a : integer;DUMMY:Textfile;
                    :
           Read (a);
```

(* Read from the file INPUT (default value)*)

Read (DUMMY, a)

(* Read from the file DUMMY *).

(a.4) Formatted parameter : Formatting of an actual parameter may be allowed.

Example : WRITE (X : I : J)

(b) Criterion of type compatibility in addition to that defined for user defined functions.

(b.1) Class equivalence :

If TYPEACTUALTAG (Defined in the Domain TYPEACTUALINFO) in the types of formal and corresponding actual parameter is same then they are class equivalent.

Example : Reset (FILET)

Where FILET should be of file type. There is no restriction over the component type of FILET.

(b.2) Relaxed Ordinal Equivalence :

Types T1 and T2 are compatible under this equivalence rule if both T1 and T2 denote the same ordinal type. The ranges of values spanned by T1 and T2, may not be the same, and, are of no consequence.

Example T.1 :

```
VAR A : INTEGER; B : 1..10;
BOOL : BOOLEAN:
  :
BOOL : = A < = B;
```

A and B are compatible types(for operator < =) as they are of same ordinal type, namely INTEGER, though A can assume any value in the set of integers whereas range of B is restricted from 1 to 10.

(b.3) Relax range Compatibility :

Types T1 and T2 are compatible under this compatibility criterion if one of the following is satisfied.

(1) T1 and T2 are ordinal types and they are compatible under RELAXED ORDINAL equivalence criterion.

(2) T1 and T2 are SET types. Base types of T1 and T2 are ordinal types and they are compatible under the criterion of RELAXED ORDINAL equivalence.

(3) T1 and T2 are the same type, that is neither a file type nor a structured type with a file component (This rule is to be interpreted recursively).

(b.4) Assignment Compatibility :

A value of type T2 is designated assignment compatible with a type T1 if any of the following statement is true :

(1) T1 is the REAL type and T2 is the INTEGER type.

(2) T1 and T2 are compatible under the criterion of RELAX RANGE compatibility.

(b.5) MEMBERSHIP equivalence :

Here formal parameter type is set of values, while actual parameter is a value. Actual parameter value is required to be member of this set.

Example - T.2

```
Type LIM = 1..3;
     REC = RECORD
             A : INTEGER;
             Case L : LIM OF
               1 : ( );
               2 : (B : REAL);
               3 : (C : CHAR)
           END;
VAR  P : ^ REC;
  :
NEW (P,2)
```

The formal parameter corresponding to second actual parameter is a set of three values i.e., 1,2 and 3. The second actual parameter is compatible with the corresponding formal parameter as it is a member of the set {1,2,3}.

Membership equivalence is invoked only in the type compatibility checking of parameters for the system defined procedures NEW and DISPOSE.

(b.6) STRGCLASS equivalence : The actual parameter is compatible with the corresponding formal parameter if it is of 'STRING type' i.e., PACKED ARRAY [1.. STRGLGTH] of CHAR, where STRGLGTH > = 1.

STRING type is allowed as valid operand type in relational operators (except 'IN') and in the system

defined procedures 'WRITE' and'WRITELN'. STRG class compatibility is invoked only in these cases.

(b.7) ARRAY class :

Types T1 and T2 are compatible under this rule if the following are satisfied.

(1) T1 and T2 are Array types.

(2) ARRAY INDEX types of T1 and T2 must be compatible under RELAXED ordinal equivalence.

(3) ARRAY COMPONENT types of T1 and T2 must be compatible under STRICT type equivalence.

ARRAY CLASS compatibility is invoked only in the type compatibility checking of parameters for the system defined procedures PACK and UNPACK.

(c) TYPE Propagation :

Sometimes the type of actual parameter must be compatible with the type (or the type derived from the type) of a parameter in the same parameter list. Therefore, PROPAGATION type of the next parameter is derived from the type of the current parameter. Derivation functions for deriving propagation types are as follows :

(c.1) TYPE PROP : PROPAGATION type is same as that of the actual parameter.

Example :
```
VAR A : SET OF 1..10; B : SET OF 5..50;
 :
A := A + B
```

Type of B must be RELAX RANGE compatible with the type of A since the operator '+' is intended to be set union.

Thus type compatibility checking of operand A with the corresponding formal parameter must synthesize a PROPAGATION type same as the type of A.

(c.2) LST Propagation :

PROPAGATION type is the list which is obtained during type matching of an actual parameter with the corresponding formal parameter. This is used only with the type compatibility checking of system defined procedures NEW and DISPOSE.

Example :

Same as T.2.

Type compatibility checking for each actual parameter of system defined procedures NEW and DISPOSE synthesizes a list of values and sends it as PROPAGATION type for the type compatibility checking of successive actual parameters. This list consists of CASE constants encountered in the variant part at certain nesting level of the parent record structure specified by the first actual parameter.

In our example type compatibility check of actual parameter P synthesizes a list consisting of values 1,2 and 3.

(c.3) CHANGE PROPAGATION

It employes two derivation functions for synthesizing propagation type from the type (T1) of actual parameter.

The derivation functions chosen on the basis of T1 are as follows :

(1) If T1 is any structured type then the PROPAGATION type is same as T1 with the packing information complemented.

(2) If T1 is any ordinal type then the PROPAGATION type is made equal to SET OF T1. It is used only in the type compatibility checking of operands for the system defined operator IN.

Example : T-3

```
VAR A : ARRAY [1..10] CHAR;
    B : PACKED ARRAY [1..5] of CHAR;
    I : INTEGER;
    :
PACK (A,I,B)
```

The actual parameter B must be ARRAY CLASS compatible with the type derived from the type of A. Thus type compatibility checking of actual parameters A must synthesize PROPAGATION type which is same as type of A except the ARRAY PACKING being changed from UNPACKED to PACKED.

(d) Operands for dyadic operators must be compatible with each other apart from being compatible with the corresponding formal parameters of aperators.

Example T.4 :

```
VAR A : SET OF 1..20; I : INTEGER;
      :
A : = A+I;
```

Although the operands A and I are compatible with the corresponding formal parameters associated with the operator '+', this operator is not defined for this combination of operands.

This conflict is resolved by employing a lattice structure of types (WAI 76, HEX 67) associated with each operator.

2.2.2. SOLUTION :

2.2.2.1 Domains : Domains chosen for type compatibility checking are discussed IN PUK 81.

2.2.2.2: We design a mechanism which allows uniform handling of operand compatibility for user defined procedures (functions) as well as PASCAL defined standard procedures (functions). Operator Identification is resolved through the use of seperate lattices for each dyadic operator that defines the possible types of each of the operands with respect to the type of the operator.

Type compatibility can be viewed as operand compatibility for various OPERATORS. An OPERATOR may be one of the following :

(1) user defined procedure

(2) user defined function

(3) System defined function

(4) System defined procedure

(5) System defined operator.

The following is the description of implementation of type compatibility checks for various OPERATORS.

(a) User defined procedures/functions

Invocation of a procedure (function) is compatible with its definition if the corresponding elements of the lists of formal and actual parameters are compatible. There are three class of parameters for these OPERATORS. Type compatibility criterion applied to them is as follows :

Class of Parameter	Type compatibility criterion applied
VAR	STRICT TYPE
VALUE	ASSIGNMENT
Procedure (function)	STRICT TYPE

(b) System defined procedures :

The type compatibility checking for user defined procedures/functions is enhanced to handle the special features present in system defined

procedures/functions or operators. As the PASCAL syntax for definition of procedures and functions does not allow the declaration of polyadic, polymorphic, default and formatted parameters, type compatibility checking for user defined procedures (functions) remains the same without any conflicts with that of system defined procedures (functions).

While enhancing the type compatibility checking for system defined procedures (functions) certain ambiguities have crept in our scheme. The following is a brief description of these ambiguities and the way they have been resolved :

(1) Default parameters :

If a formal default parameter and its successive formal parameter are of the same type, then our scheme for type compatibility checking cannot decide unambiguously the presence or absence of actual parameter corresponding to formal default parameter.

In all system defined procedures (functions) a default parameter is never of the same type as its successive parameter.

(2) Formatted parameter : While processing an expression in the actual parameter list of a system defined procedure, our scheme requires that we distinguish whether this expression is a new actual parameter or whether it denotes formatting information about

the previous actual parameter. This is resolved through PASCAL syntax in specifying format of an actual parameter, i.e., two actual parameters are delimited by 'comma' whereas an actual parameter and its format are seperated by 'colon'.

(c) System defined functions and operators

In this section we discuss the special feature required for type compatibility checking of system defined functions/operators and the way they have been implemented in our scheme.

1. Operator Identification

An operator symbol may stand for more than one meaning depending upon type of its operands. Operator identification is discrimination between the various meanings of an overloaded operator symbol.

Each operator is associated with a lattice of types [HEX 67]. Each node in this lattice has an associated "lattice value".

Type compatibility checking of each operand of an operator also generates a lattice value. After establishing compatibility of two successive operands of an operator with the corresponding formal parameters, the pair of lattice values generated are used to traverse the lattice associated with this operator. This traversal returns a lattice value which is a least upper bound of the input lattice values.

The resultant lattice value conveys whether the combination of the operand types is valid for this operator.

If the operand types are compatible with the corresponding formal parameters and the combination of operand types is meaningful for this overloaded operator then the particular meaning assigned to the operator is found out from propagation type.

2. Result type

The result type of system defined functions and operators may be polymorphic. Theoretically, the resultant lattice value, generated after processing the operand list, uniquely determines the result type of the operator.

In our implemention we have not chosen an explicit lattice structure for each operator. Only four different lattice values are generated by type compatibility checking of operands. The lattice traversal routine, namely 'JOIN', takes in combination of two such lattice values and Propagation type. It returns the resultant type of the operator and uniquely identifies an operator.

The above scheme to handle type compatibility checking for system defined functions and operators does not conflict with that of user defined functions

(procedures). The lattice value zero (0) is always generated, while performing type compatibility checks of actual parameters of user defined procedures (functions). In this case the resultant lattice value generated by 'JOIN' is also '0' and result type for functions is found out from the definition of function name in symbol table.

CHAPTER 3

LABEL PROCESSING

3.1 Structure of the Problem :

Labels and GOTOs satisfy the following properties

(1) The scope of a label is the BLOCK in which it is declared.

(2) A label is defined by prefixing it to a statement (nest) and is considered as local to the nest within which this statement occurs.

(3) A label is visible only in the nest within which it is defined.

3.2 Solution :

The solution we present handles all problems concerning local labels, i.e., their definition and check for valid reference. With regard to global GOTOs, we restrict the problem to the definition of such labels in the outer most nest of their respective blocks.

To solve this problem through L-EAG we have chosen following attribute domains.

(1) LIST OF LABELS : Two attributes of this domain are DECLABELS and DEFLABELS. DECLABELS keeps the list of labels declared in a block. DEFLABELS keeps the list of labels defined in the block.

(2) LABELS-TO-JUMP : A list of forward referenced labels along with the level of the nest in which it is used.

(3) STACK OF LABELS : It contains those defined labels which are visible in a nest. The level of stack denotes the level of the nest in which the associated labels are defined.

(4) STACK OF DECLABELS : It is the stack of DECLABELS which are accessible for referencing in the current block.

The actions taken at different instances for solving this problem are as follows :

(1) On definition of a label :

(a) The label must be local to the current block. This is fulfilled by ascertaining that this label belongs to DECLABELS, i.e., <where label ϵ DECLABELS>.

(b) Unique Definition :

Disjoint union of DEFLABELS with the label is taken, i.e., DEFLABELS : = DEFLABELS $\not\emptyset$ [label].

(c) Validity of forward reference :

Forward references are detected by the presence of the label in LABELS-TO-JUMP. For an instance of the label in LABELS-TO-JUMP, its associated level value should be equal to the level of the current nest. All forward references corresponding to this label are delected from LABELS-TO-JUMP.

(2) On reference to a label :

(a) Scope check : The label must be in the STACK OF DECLABELS.

(b) If the label belongs to DEFLABELS then it must belongs to STACK OF LABELS thereby satisfying the visibility criterion.

(c) If it is a forward reference, i.e., the label does not belong to DEFLABELS, then it is put in the LABELS-TO-JUMP with the information of the level of nest.

(3) On END OF Nest

The level information associated with each label forward referenced in this nest is set to the nesting level of the enclosing nest.

(4) On start of nest :

The level of nesting is increased by 1 on entry to any nest, and is passed as inherited attribute to the nest.

If the nest is a statement sequence, the level of nesting passed to it is the same as that of enclosing nest. If the nest is a compound statement, and the enclosing nest is the statement part of a block, then again the same nesting level is passed.

(5) On end of scope :

(a) All labels declared in the current block must have been defined, i.e., they must belong to DEFLABELS. (Warning is given if a declared label does not belong to DEFLABELS).

(b) All labels declared in the current block are delected from LABELS-TO-JUMP.

(6) At the end of procedure (function) declaration : LABELS-TO-JUMP obtained as the synthesized attribute from the procedure (function) declaration is appended to the LABELS-TO-JUMP list of the current block.

CHAPTER 4

ALIASING

4.1 Structure of the problem :

Aliasing can be thought of as an object having multiple names. We are concerned with aliasing of variable. Thus for our purpose aliasing is the referencing of a memory location by more than one name. Thus memory location contents can be modified by assignment to its intended name as well as to one or more of its aliased names. This may result in unexpected changes in the value of a variable.

Here we design a scheme which allows modification of contents of memory location only through assignment to its intended name. The intended name is taken as the one which is declared closest to the point of modification of the value of the variable.

Causes of Aliasing :

Aliasing may arise in the following cases :

(a) When a variable which is global to the called procedure is passed as actual VAR parameter to this procedure, in the block of called procedure both names refer to the same memory location.

(b) When the same variable is passed as two actual VAR parameters, in the block of called procedure the corresponding formal parameters refer to the same memory location.

4.2 Solution

Various domains used in solving the problem of aliasing are as follows :

(1) LSTOFLST : Attribute variables GLOBALVARLST and REFLST belong to this domain. GLOBALVARLST contains list of all the global variables modified in the current block. REFLST is the list of the actual VAR parameters which are global to the called procedure.

(2) QUALIFYLST : Attribute variable REFLSTELEM belongs to this domain. It contains a variable which is an element of GLOBALVARLST or REFLST.

(3) REFTABLEINFO : REFTABLE is the attribute variable belonging to this domain. It contains the information about aliasing of all the procedures defined and declared in the current Block.

(4) STACK OF REFTABLEINFO : It keeps the information about aliasing of all the procedures which are visible from the current Block.

To recognise aliasing the following actions are performed at different points of parsing.

(1) On encountering an assignment statement :

(a) If the current block is a function block then no action is taken as we attempt to check that functions are side-effect free.

(b) The current block is a procedure block and the variable on the L.H.S. of an assignment statement is global to the Block :

Prepare a REFLSTELEM consisting of the qualified name of the variable and append it to GLOBALVARLST.

(2) On encountering a procedure call :

(a) If the block of the called procedure is already declared then the actions taken are :

(i) Find GLOBALVARLST of the called procedure from the STACK OF REFTABLEINFO.

(ii) For each actual VAR parameter which is global to the called procedure the following action is taken :

REFLSTELEM is prepared for this parameter. We take intersection of REFLSTELEM and GLOBALVARLST (found in Step (i)). The intersection should be nil for the case of no aliasing.

(b) If the block of the called procedure is not yet declared then the following actions are taken :

(i) Prepare REFLST for all the VAR parameters which are global to the called procedure.

(ii) Put REFLST in the stack of REFTABLEINFO associated with the called procedure name.

(3) At the end of a procedure block :

In case there have been calls to this procedure before its body was processed then the following actions are taken:

(a) The REFLST associated with the procedure name is found out from the stack of REFTABLEINFO.

(b) We take the intersection of this REFLST with the GLOBALVARLST prepared in the body of this procedure. Variables common to the two lists are aliased names.

(c) The GLOBALVARLST prepared in this block is kept with the procedure name in the stack of REFTABLEINFO.

In addition, a list of all actual var parameters is prepared on the call of a procedure so as to detect duplication of VAR parameters.

This scheme does not recognise aliasing due to propagation which may occur because of the following reasons :

(1) When a VAR formal parameter is passed as a VAR actual parameter to an enclosed procedure block.

(2) When a procedure or function is passed as an actual parameter.

CHAPTER 5

SPECIAL CASES

This chapter briefly presents some other special cases we have come across while performing context sensitive analysis in PASCAL programs. These problems are not well formalized and they are mainly pertain to our implementation.

5.1. Type compatibility checks for recursively defined data structures :

Example

```
Type Node1  = ↑ Node3;
     Node2  = ↑ Node4;
     Name   = Packed Array [1..5] of char;
     Node3  = Record
                Nodname : Name;
                LPTR    : Node1
              END;
     Node4  = Record
                Name1 : Name;
                Nextptr:Node2
                END;
Var Test1 : Node1; Test2 : Node2;
  :
Test1 : = Test2
```

The normal scheme of type compatibility checking of Test1 and Test2 requires type compatibility of Node1 and Node2

which further requires type compatibility of Node3 and Node4. This in turn requires type compatibility of Node1 and Node2 resulting in a cyclic process with no termination.

We modify the type compatibility checking as follows : Attribute domains chosen for this purpose are

a) SAFE : SAFELST is an attribute variable of this domain.

b) PTRENVIRONMENT : ADDRESS1 and ADDRESS2 are attribute variables of this domain.

These domains are utilised in our scheme to tackle the above example as follows :

The type compatibility checking of Node1 and Node2 is done as follows :

Since both are pointer types the address information is taken out from the type domain of node1 and node2. Their addresses are ADDRESS (Node3) and ADDRESS (Node4). This tuple of addresses is searched in SAFELST. If present then they are assumed to be compatible. Otherwise that tuple is appended to the SAFELST.

5.2 Illegal use of control variable of a for loop within the loop.

A control variable used in the for statement shall fulfill the following criterions :

(1) The variable must have been declared in the VAR declaration part of the current block.

(2) No assigning reference to control variable shall be made within the for loop.

4.2.1 Solution

The KIND attribute of all the variables declared in VAR declaration part is VARR. The level of the block in which the variable is declared is also stored along with the variable. Therefore the KIND attribute of for loop variable will be VARR and its level must be same as that of current block. At the start of for loop, KIND of control variable is changed to CONTVAR. Assigning reference to a variable of KIND CONTVAR is not allowed. At the end of for loop the attribute KIND of the control variable is again changed to VARR, thereby allowing assigning reference to it beyond the scope of the loop.

5.3 The illegal use of with record variable inside with statement :

5.3.1 Structure of problem :

With record variables must not be modified inside with statement, as it may result in undesirable effects.

Example

```
Type    Nodeptr = ↑ Node;
        Node    = Record
                      name : Array [1..10] of char;
                      LPTR, RPTR : Nodeptr
                  end;
var     p,q : Nodeptr ; FLAG : Boolean;
  :
```

```
with p ↑ do begin
            FLAG : = TRUE;
            while (p ≠ NIL) AND FLAG do begin
                  If NAME ≠ 'XY          ', then FLAG: = FALSE;
                  P: = RPTR
                  end;
```

In this program, intention is to traverse a list headed initially by 'p'. But it results in infinite looping because 'RPTR' is always the same i.e. 'RPTR' of head of list. The local environment accessible inside a with statement remains unchanged even though an assigning reference to with record variable is made. Therefore any such attempt to change the local environment must be detected and properly warned.

5.3.2 Solution :

Domains used for this purpose are :

(a) LSTOFLST : WITHLST is an attribute belonging to this domain. It is a list of all the record variables which control the local environment accessible inside with statement.

(b) PTRQUALIFYLST : WITHLSTELEM is an attribute of this domain. While processing the assigning reference to a variable WITHLSTELEM stores the qualified name of that variable.

Actions taken during different points of parse are:

(1) On start of with statement :

WITHLST is extended by with record variables which specify the local environment accessible to the nest to follow.

(2) On assigning reference to a variable :

A WITHLSTELEM is prepared and an instance of this is searched in the WITHLST. If search succeeds then error message is given.

(3) On end of with statement :

All the with record variables which governed the current nest are deleted from the WITHLLST.

CHAPTER VI

EPILOGUE

The aim of this project was to obtain a formal basis for the specification of the static context in PASCAL programs and provide a basis for the conversion of the specification to program. We have been motivated to use L-EAGs because of their adaptability to LL(1) parsing : LL(1) provides a general basis for the structure of the program, in which is embedded the checks of static context obtained through L-EAG specification.

Our experience is that the articulation of the L-EAG specification helped clarify several deep issues in the programming language PASCAL. The L-EAG used proved to be extremely concise at the same time completely defining the the context-free and context-sensitive syntax of Pascal. The underlying context free syntax was clearly visible. The conciseness stems mainly from the freedom to choose any suitable domains and operations for the attributes, and the ability to use attribute expressions freely in the semantic rules.

With regards to rules of transformation which convert L-EAG specification to Pascal programs, the following observations are in order :

(a) Proper use of nesting of generated procedures can help in increasing the efficiency of the resultant program.

(b) Aids to perform L-EAG to L-EAG transformation so that the resultant specification is efficient with regards to its utilization of attributes would be of great value.

(c) The transformation rules that we have given pertain to Attribute Grammar and they need careful adaptation to L-EAG.

The articulation of L-EAG specification, transformation rules and Pragmatic Transformation Strategy used greatly facilitated in handcoding the program with confidence. The following observations are made in coding of this program :

(1) Most of the bugs in the resulting program originated from the improper initialization of POINTERS which were elements of newly generated objects of various domains. In retrospect, it would prove fruitful to write an allocation procedure for every dynamically creatiable domain element so that the component pointers can be appropriately initialized.

(2) Use of identifier names too close to the standard names in PASCAL should be avoided as far as possible. For example we have used ORD as a field name in a record structure. Thus inside a with statement, wherein the local environment of this record can be accessed without qualification, the standard function ORD cannot be used.

(3) We have used files as a way out of the garbage collection problem whenever the end of a scope is encountered. This is because of forward references to objects yet not defined within the scope. This implementation strategy can be avoided if all newly allocated domain objects are linked with respect to their logical use. Garbage collection will merely involve the recycling of lists no longer needed. Proper implementation of this strategy would require central allocation and deallocation routines for domain elements.

(4) The ISO standard for PASCAL tightens several context-sensitive constraints imposed on a program. Almost all Pascal programs of any consequence do not satisfy the constraints imposed by ISO standard. The commonly detected errors by our processors are :

(a) Improper uses of labels and goto's

(b) Modification of with RECORD variables encountered in WITHLIST

(c) Aliasing

(d) Use of globally declared variables as control variables in FOR STMT.

We feel that some of these errors are reported because of the strictness of the constraints and not because of poor programming practice.

Finally though we have derived L-EAG for PASCAL and our observations regarding transformation rules etc., are specific to PASCAL, we feel that very similar strategies and much of what we have discussed is applicable to contemporary 'typed' programming languages.

References

1. SIG80 Addyman, A.N. : A Draft Proposal for PASCAL. ACM SIGPLAN Notices 15, 4 (April 80)

2. PUK81 Kachhwaha, P. : Synthesis of Static Context in PASCAL Programs with L-Attribute Grammars. M.Tech. Thesis, IIT Kanpur, 1981.

3. WAT79 WATT, D.A. : An Extended Attribute Grammar for PASCAL. ACM SIGPLAN Notices 14, 2 (Feb 79).

4. WM77 WATT, D.A. and Madsen, O.L. : Extended Attribute Grammars. Report No. 10 July 1977.

5. HEX67 Hext, J.B. : Compile Time Type Matching. Computer Journal 1967.

6. WAI76 WAITE, W.M. : Semantic Analyser. In Compiler Construction : An Advanced Course, Ed. by Bauer, F.L., Eickel, J., Springer-Verlag, NV, 1976.

APPENDIX -I

DOMAIN DEFINITION

Domains used in L-EAG specification of standard PASCAL are described below.

```
SYMBOLTABLE =     stack of
                  (blockinfo:BLKINFO;symenviron:ENVIRON;
                   reftabinfo:REFTABLEINFO;
                   labelinfo:LABINFO)

BLKINFO =         (blkname:BLKNAMETYPE;blktype:BLOCKTYPE)

BLKNAMETYPE =     (blktype:BLOCKTYPE;
                  (maintype(nameofblk:NAME)|
                   proctype(nameofblk:NAME)|
                   functype(nameofblk:NAME)|
                   recordtype(recnamelst:QUALIFYLST))

BLOCKTYPE =       (maintype,proctype,functype,recordtype)
                              *
QUALIFYLST =      QUALLSTELEM

QUALLSTELEM =     (ofnamevaltag:NAMEVALTYPE;
                  (valueconst:(qualvalue:INTEGER)|
                   nameconst:(qualname:NAME))

NAMEVALTYPE =     (valueconst,nameconst)

ENVIRON =         < NAME-->MODE >

MODE =            (modekind:KIND;modetype:TYPE;
                   idlevel:INTEGER)
```

```
KIND =              (kindtag:KINDTAGTYPE;
                    (void|
                     const(kindconstval:CONSTVALTYPE)|
                     varr|
                     type|
                     func(plan:PLAN)|
                     localfunc(plan:PLAN)|
                     formalfunc(plan:PLAN)|
                     proc(procplan:PLAN)|
                     formalproc(procplan:PLAN)|
                     forwardfunc(funcplan:PLAN;funcenv:ENVIRON)|
                     forwardproc(procplan:PLAN;procenv:ENVIRON)|
                     field:(kindtaglst:TAGLSTTYPE)|
                     formalvar|
                     formalval|
                     boundid|
                     contvar|
                     exprkind))

CONSTVALTYPE =      (constvaltag:VALUETAGTYPE;
                    (valuepresent:(value:INTEGER)|
                     valueabsent))
                              *
PLAN =              TYPEACTUAL
                                                          *
TAGLSTTYPE =        (tagname:NAME;tagcasevalue:VALLSTTYPE)
                          *
VALLSTTYPE =        INTEGER

TYPE =              (typeact:TYPEACTUAL;fileselect:BOOLEAN;
                     fatherpacking:PACKINGTYPE)

PACKINGTYPE =       (unpacked,packed)
```

```
TYPEACTUAL =     (typeactualtag:ACTUALTAGTYPE;
                 (void!
                  real!
                  ordinal(ord:ORDTYPE;ranse:RANGETYPE)!
                  arrayy(packins:PACKINGTYPE;
                         indextype:TYPEACTUAL;
                         elementtype:TYPE)!
                  confarray(packins:PACKINGTYPE;
                         confindextype:TYPEACTUAL;
                         elementtype:TYPE)!
                  recordd(packins:PACKINGTYPE;
                         layout:PLAN;recenviron:
                         ENVIRON;recbtmptr:BTMPTRTYPE)!
                  union(discrimination:DISCTYPE;
                         tastype:TYPEACTUAL)!
                  sett(basetype:TYPE)!
                  nullset!
                  filee(comptype:TYPE)!
                  textfilee!
                  pointer(addr:ENVIRON)!
                  nilptr!
                  formalparconst(formalparvar:FORMALPARTYPE)))

ORDTYPE  =       (ordtag:ORDTAGTYPE;(int!char!boolean!
                 enum(enumlist:NAMELSTTYPE;enumcount:
                 INTEGER)))
                    *
NAMELSTTYPE =    NAME

RANGETYPE =      (ransetag:RANGETAGTYPE;(all!
                 subranse(franse:INTEGER;lranse:INTEGER)!
                 valued))

BTMPTRTYPE =     (tastypeactual:TYPEACTUAL;varaccesslst:
                 VALUEPTRLST)
                                                    *
VALUEPTRLST =    (value:INETGER;btmptr:BTMPTRTYPE)

DISCTYPE =       (discriminated;free)

FORMALPARTYPE =  (partag:PARTAGTYPE;
                 (simplepar(simple:SIMPLETYPE)!
                  defaultpar(comsimple:COMSIMPLSTYPE)!
                  polymorph(comsimple:COMSIMPLSTYPE)!
                  polyadic(comsimple:COMSIMPLSTYPE)))
```

```
                                        *
COMSIMPLSTTYPE = COMPSIMPLETYPE
COMSIMPLETYPE = (comsimptag:COMSIMPTAGTYPE)
                (verysimple(simple:SIMPLETYPE)!
                 formatted(formatsimple:SIMPLSTTYPE))
                          *
SIMPLSTTYPE =    SIMPLETYPE

SIMPLETYPE =     (lastpropogated:BOOLEAN;criteria:CRITERION;
                  nextprop:NEXTPROPAGATION;simplemode:MODE;
                  latticevalue:INTEGER)

CRITERION =      (typeequivalence,assignment,membership,
                  class,ransferelax,arrayclass,strclass,
                  arrransferelax)

REFTABLEINFO =   { PROCINFO --> PROCMODETYPE }

PROCINFO = (nameofproc:NAME;levelofproc:INTEGER)


PROCMODETYPE =   (reflst:LSTOFLST;globlst:LSTOFLST;
                  procblkdectag:BOOLEAN)
                          *
LSTOFLST =       QUALIFYLST
                      *
LABINFO =        INTEGER
                    *
MODELSTTYPE =    MODE

STACKLBLTYPE =   { INTEGER-->LBLLSTTYPE }
                      *
LBLLSTTYPE =     INTEGER
                       *
ADRLST =         ENVIRON

SAFETYPE =       { addr1:ENVIRON;addr2:ENVIRON }
                                                        *
LABJUMPTYPE =    (labelvalue:INTEGER;levelvalue:INTEGER)
```

APPENDIX 2

The following is a list of attribute variables used in the L-EAG productions together with their domains:

SYMBOL TABLE : Globaltable, Symboltable, Symboltable1, Symboltableout, Symboltablein, Symboltablemid.

BLKINFO: Blockinfo

QUALIFYLST : Globallistelement, Reflistelement

QUALSTELEM : Loclstelem, curlstelem

ENVIRON : Localtable, localtableconst, localtableconst1, Localtableconst2, Tabledefin, Tabledefout, Tabletype, Tabletypeindex, Recordtablein, Recordtableout, Recordtablesection, Tabletype1, Recordtablevariant, Tabletypeout, Proctablein, Proctable, Proctableout, Proctable1, Proctablemid, Proctable2, Localtablehead, Address, Localtablevar, Localtableproc, Functable.

MODE : Mode, Expmode, Paramode, Simpmode

KIND : Kind, Modekind, Calledprockind, Parentkind, Parameterkind

KINDTAGTYPE : Kindtag, Kind1

PLAN : Layout, Layin, Layoutsection, Plan, Planin, Planout, Formalactualtypelist, Remformalactualtypelist Formaltypeactuallistin, Formaltypeactuallistout

TAGLSTTYPE : Taglist, Taglistout, Taglistl

TAGLSTELEM : Taglstelement

VALLSTTYPE : Taglistvalue, Taglistvaluel, Caseconstlistl, Caseconstantlist, caseconstlist2, Tagvaluelist, Taglistvaluein, Taglistvalueout

TYPE : Type, componentype, comptype, filedomain, domaintype, elementtype, recordtype

PACKINGTYPE: Parentpacking, packing, selfpacking

TYPEACTUAL : Consttype, Actualtypearray, Indextype, Actualtype, Tagtypeactual, Typeactual, Resulttype, Propagatedtypeactual, Propagationtypeactual, Propagationtypeactualmid, Typeactuall, Typeactual2

ORDTYPE : Ordinal, Ordinall, Ordinal2

NAMELSTTYPE: Filestocheck, Remfiles, Identifierlist, Namelist, Remfilestocheck, Parameterlist

RANGETYPE : Rangel, Range2

BTMPTRTYPE : Btmptr

VALUEPTRLST: Variantaccesslist, variantaccesslistin

DISCTYPE : Discrimination

FORMALPARTYPE : Formalparvar

COMSIMPLSTTYPE: Combinedsimplelist

COMSIMPLETYPE : Combinedsimple, combinedsimplel, combined-simple2

SIMPLSTTYPE : Simplelist

SIMPLETYPE : Simple, Simplel, Simple2

CRITERION : Criterion

REFTABLEINFO : Reftableblock, Reftableout, Reftablein, Reftableoutl, Reftableout2

REFTABLEELEM : Reftableelementin, Reftableelementout

LSTOFLST : Reflst, Globlist, Globvarlstl, Globvarlst2, Globvarlstmid, Withlist, Globalvarlist, Globallist, Withlistl, Withlist2, Withlistmid Parameterlist, Parameterlistmid, Varlistin, Varlistmid

MODELSTTYPE : Actualmodes

STACKLBLTYPE : Globallabels, Stackoflabels,stackoflabelsl, Stackoflabelsin

LBLLSTTYPE : Locallabels, Decllables, Decllabelsl, Decllables2, Decllabelsin, Decllablesout

ADRLIST : Ptrnames, Ptrnamesdef, Ptrnamesout, Ptrnamessection

SAFETYPE	: Safe1, Safe2, Safemid, Safe3
LABELJUMPTYPE	: Labelstojump, Labelstojump1, Labelstojump2, Labelstojumpin, Labelstojumpout
NAME	: Procname, Funcname, Name, Idname
BOOLEAN	: Typevarflag, Fileselect, Fixpart, Formaltag, Forwardflag, Presenceflag, Withtag, Varaccesstag, Errinexp, Mismatch, Exitflag, Errsuppress
INTEGER	: Level, Blocklevel, Withblklevel, Nestinglevel, Procblocklevel, Calledproclevel, Latticevalue

-

APPENDIX III

The following is the list of L-ATG productions for the programming language PASCAL. These productions are pertaining to declaration part of the PASCAL programs.

The following conventions have been adopted in the following grammar text:

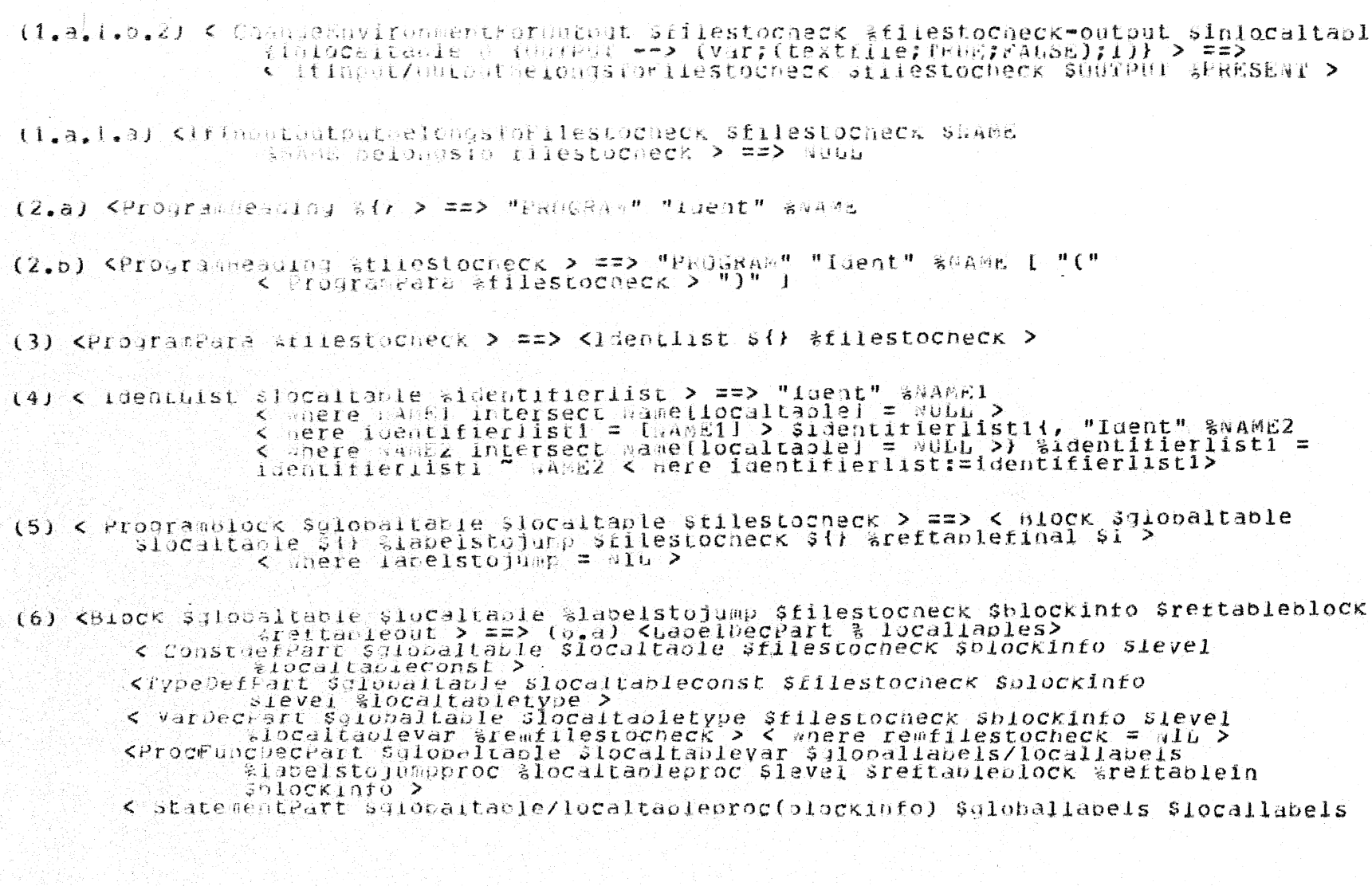
- "$" stands for inherited attribute
- "%" stands for synthesised attribute
- "^" stands for append (concatenation)
- "~" stands for disjoint append
- "@" stands for disjoint union
- "V" stands for logical OR
- "&" stands for logical AND
- "U" stands for set union

(1) < Program > ==> < Programheading %filestocheck > ";" < Prepareenvironment %globaltable >
<Changeenvironmentforinputoutput $filestocheck %remfiles %localtable >
< Programblock $globaltable $localtable $remfiles >

(1.a) <Changeenvironmentforinputoutput $filestocheck %remfiles %localtable > ==>
< ifinput/outputbelongstofilestocheck $filestocheck $input %ABSENT >
< Changeenvironmentforoutput $filestocheck %remfiles $1) %localtable >

(1.a.2) < Changeenvironmentforinputoutput $filestocheck %remfiles %localtable > ==>
<ifinput/outputbelongstofilestocheck $filestocheck $INPUT %PRESENT >
<Changeenvironmentforoutput $filestocheck - [INPUT] %remfiles
%[INPUT --> (var;(textfile;TRUE;FALSE); 1)] %localtable >

(1.a.1.b) <Changeenvironmentforoutput $filestocheck %filestocheck $inlocaltable %inlocaltable >
<ifinput/outputbelongstofilestocheck $filestocheck $OUTPUT $ABSENT >


(1.a.i.b.2) < ConstEnvironmentForInput $filestocheck %filestocheck-output $inlocaltable
$inlocaltable = INPUT --> (Var;(textfile;PROG;FALSE);1)} > ==>
< ifinput/outputbelongsforfilestocheck $filestocheck $INPUT %PRESENT >
(1.a.i.a) <ifinputoutputbelongsforfilestocheck $filestocheck $NAME
%NAME belongsto filestocheck > ==> NULL
(2.a) <Programheading %{} > ==> "PROGRAM" "Ident" %NAME
(2.b) <Programheading $filestocheck > ==> "PROGRAM" "Ident" %NAME ["("
< ProgramPara %filestocheck > ")"]
(3) <ProgramPara $filestocheck > ==> <identlist ${} %filestocheck >
(4) < identlist $localtable %identifierlist > ==> "Ident" %NAME1
< where NAME1 intersect name(localtable) = NULL >
< here identifierlist1 = [,NAME1] > $identifierlist1{, "Ident" %NAME2
< where NAME2 intersect name(localtable) = NULL >} %identifierlist1 =
identifierlist1 ~ NAME2 < here identifierlist:=identifierlist1>
(5) < ProgramBlock $globaltable $localtable $filestocheck > ==> < Block $globaltable
$localtable ${} %labelstojump $filestocheck ${} %reftablefinal $i >
< where labelstojump = NIL >
(6) <Block $globaltable $localtable %labelstojump $filestocheck $blockinto $reftableblock
%reftableout > ==> (6.a) <LabelDecPart % locallables>
6.b) < ConstDefPart $globaltable $localtable $filestocheck $blockinfo $level
%localtableconst >
6.c) <TypeDefPart $globaltable $localtableconst $filestocheck $blockinfo
$level %localtabletype >
.d) < VarDecPart $globaltable $localtabletype $filestocheck $blockinfo $level
%localtablevar %remfilestocheck > < where remfilestocheck = NIL >
.e) <ProcFuncDecPart $globaltable $localtablevar $globallabels/locallabels
%labelstojumpproc %localtableproc $level $reftableblock %reftablein
$blockinfo >
.f) < StatementPart $globaltable/localtableproc(blockinfo) $globallabels $locallabels

```
                $labelstolumproc %labelstolumpout $blockinfo $level $reftablein
                $reftableout > < where labelstolumn intersect localllabels = NULL >

(7.a) < LabelSetPart $1? > ==> NULL

(7.b) < LabelSetPart $localllabels > ==> "LABEL" < Label1 $labelvalue>
                < where labellist1 = {labelvalue} > $labellist1{ "," <Label1 %labelvalue2 >
                / $labellist1 = labellist1 ~ {labelvalue2} < where localllabels=labellist1>

(8) < Label1 $INTVALUE > ==> "intconst" $INTVALUE < where 0<= INTVALUE <= 9999 >
                        CONSTDEFPART
                        ------------

<9.a) < ConstDefPart $globaltable $localtable $fielstocheck $blockinfo $level
                %localtable > ==> NULL

(9.b) < ConstDefPart $globaltable $localtable $filestocheck $blockinfo $level
                %localtableconst > ==> "CONST" < Here localtableconst1:=localtable >
                $localtableconst1{ < ConstDef $globaltable $localtableconst1 $filestocheck
                $blockinfo $level %localtableconst2 > ";" }+ %localtableconst1:=
                localtableconst2 <here localtableconst:=localtableconst1 >

(10) <ConstDef $globaltable $localtableconst $filestocheck $blockinfo $level
                %localtableconst # {NAME ==> (const;(consttype;FALSE;UNPACKED);level)}>
                ==> "ident" %NAME <where NAME does not belong to filestocheck >
                "=" < Constant $globaltable/localtableconst(blockinfo) %consttype >

(11.a) < Constant $symboltable %(ORDINAL(INT;VALUED(INTVALUE)) > ==> "intconst" %INTVALUE

(11.b) <Constant $symboltable %(ORDINAL(INT;VALUED(OP*INTVALUE)) ==> "Sign" %OP
                "intconst" %INTVALUE

(11.c) < Constant $symboltable %real > ==> "realconst"
```

```
(11.d) < Constant $symbolTable $REAL > ==> "Sign" $DP "Realconst"

(11.e) < Constant $symbolTable %constType > ==> < ConstId $symbolTable %constType >

(11.f) < Constant $symbolTable %(ORDINAL(INT;VALUED(DP*value) > ==> "Sign" $DP
                 <Constid $symbolTable $ORDINAL(INT;VALUED(value)) >

(11.g) < Constant $symbolTable %REAL > ==> "Sign" $DP <Constid $symbolTable %REAL >

(11.h) <Constant $symbolTable %ARRAY(PACKED;ORDINAL(INT;SUBRANGE(1;STRGLGTH));
                 (ORDINAL(CHAR;ABS));FALSE;TRUE)) ==> "Strnconst" %STRGLGTH

(11.i) < Constant $symbolTable $ORDINAL(CHAR;VALUED(ORDVALUE)) > ==> "Charconst" %ORDVALUE

(12) < Constid $symbolTable %constmode.type.typeactual > ==> "Ident" %NAME
                 < Here constmode = mode(NAME$symbolTable) >
                 <where constmode.kind.kindtag=constt>

(13.a) < TypeDefPart $globalTable $localTableconst $filestocheck $blockinfo $level
                 %localTableconst > ==> NOOP

(13.b) < TypeDefPart $globalTable $localTableconst $filestocheck $blockinfo $level
                 %localTabletype > ==> "TYPE" < here tabledefin :=localTableconst
                 and ptrnames:=[] > $ptrnames $tabledefin1 < TypeDef $globalTable
                 $tabledefin $filestocheck $blockinfo $level %tabledefout $ptrnames
                 %ptrnamesout >; %tabledefin:=tabledefout %ptrnames:=ptrnamesout
                 < here localTabletype:=tabledefin >
                 <CheckForPtrNames $globalTable/localTable $ptrnames >

(14) < TypeDef $globalTable $localTable $filestocheck $blockinfo $level %tabletype1
                 $ptrnames %ptrnamesdef > ==> "Ident" %NAME <where NAME does not belong
                 to filestocheck > "=" < TypeDenoter $globalTable $localTable $filestocheck
                 $blockinfo $level %tabletype $[] %ptrnamesdef SUBPACKED %type $TRUE >
```

```
                    ; does address belong to ptrnames
                    TRUE:<where address:=address(NAME;globaltable/localtable(blockinfo))>
                         <where tabletype:=localtable and type:=address.mode.type and
                          ptrnamesset:=ptrnamesset-[NAME] >
                    FALSE:< where tabletype:=tabletype & {(name ==>(TYPE;type;level))} >

(15) < TypeDenoter $globaltable $localtable $filestocheck $blockinfo $level %tabletype
       $ptrnames %ptrnamesdef $parentpacking %type $typevarflag > ==>
       <SimpleType $globaltable $localtable $filestocheck $blockinfo $level
       %tabletype %type $ptrnames $typevarflag > | <StructuredType $globaltable
       $localtable $filestocheck $blockinfo $level %tabletype $ptrnames
       %ptrnamesdef $parentpacking %type $typevarflag > |
       <PointerType $globaltable $localtable $localtable $parentpacking
       %type $ptrnames %ptrnamesdef $typevarflag %tabletype $level >

(16.a) <SimpleType $globaltable $localtable $filestocheck $blockinfo $level %localtable
       $parentpacking %type $ptrnames $typevarflag > ==>
       <SubrangeType $globaltable/localtable(blockinfo) $parentpacking %type
       $ptrnames $typevarflag > | <SubrangeType $globaltable/localtable(blockinfo)
       $parentpacking %type >

(16.b) < SimpleType $globaltable $localtable $filestocheck $blockinfo $level %tabletype
       $parentpacking %type $ptrnames $typevarflag > ==> <EnumerationType
       $globaltable $localtable $filestocheck $blockinfo $level %tabletype
       $parentpacking %type >

(17.a) <IdBeginType $symboltable $parentpacking %(typemode.type.typeactual;
       typemode.type.fileselect;parentpacking) > ==> "Ident" %NAME
       %typevarflag
                TRUE:<where NAME does not belong to ptrnames >
               FALSE:NULL
       < here typemode:=mode(ENV[NAME])> <where typemode.kind.kindtag=TYPE >

(17.b) <IdBeginType $symboltable $parentpacking %(ORDINAL(typeactual1.ordselect;SUBRANGE
       (mode.type.typeactual.rangeselect;typeactual1.rangeselect));FALSE;
       parentpacking) $ptrnames $typevarflag > ==> "Ident" %NAME < here
       mode:=ENV[NAME] > <where mode.kind.kindtag=constt and mode.type.typeactualtag
       =ORDINAL > ".." <Constant $symboltable %typeactual1 >
```

```
                    <where typeactual1.typeactualtag=ORDINAL> <OrdinalEquivalence
                    $mode.type.typeactual.oroselect stypeactual1.oroselect %ismatch $FALSE >
                    <where mode.type.typeactual.rangeselect <=typeactual1.rangeselect >

18.   <EnumerationType $GlobalTable $LocalTable $FilestoCheck $BlockInfo $Level
      %TableType $ParentPacking $(ordinal(EnumNameList);All);false;
      ParentPacking)> ==> "(" <IdentList $LocalTable &NameList>)<WhereNameList
      intersect  "FilestoCheck =null)>
                 <EnumerationConstant $LocalTable $BlockInfo $Level %TableType $NameList
                                      $(end(namelist)) $Count>

18.a.1 <EnumerationConstant $LocalTable $BlockInfo $Level $LocalTable $[] $Ordinal
       %n ==> NOL

18.a.2 <EnumerationConstant $LocalTable $BlockInfo $Level $LocalTable  U (Name
       Level); $Name^NameList $Ordinal $0 ==> Name belongsto FilestoCheck
       true:PrepareIdentIncrementCount false:DonotIncrementCount
       <EnumerationConstant $LocalTable $BlockInfo $Level %LocalTable1 $NameList
        $Ordinal $N+1>

19.   <SubRangeType $SymbolTable $ParentPacking %(ordinal(actualtype1.ord,subrange(actualtype1.range
                    actualtype2.range));false;void;parentpacking)>
            ==> <Constant $SymbolTable %actualtype1  > ' '
          <Constant $SymbolTable %actualtype1 >
       <where ActualType1.ActualTypeTag = Ordinal><Constant $SymbolTable
       %ActualType2><where ActualType2.ActualTypeTag = Ordinal><RelaxedOrdinal
       Equivalence $ActualType1.OroSelect $ActualType2.OrdSelect> $Mismatch>
       <where ActualType1.RangeSelect <= ActualType2.RangeSelect>

20.   <StructuredType $GlobalTable $LocalTable $FilestoCheck $BlockInfo
       $Level %TableType $PtrNames %PtrNamesDef $ParentPacking %TYPE"
       ==> <HerePacking:=unpacked><packed $Packing1<UnpackedStructType
       $GlobalTable $LocalTable $FilestoCheck $BlockInfo $Level %TableType
       $PtrNames %PtrNamesDef $ParentPacking $Packing %TYPE> $TypeVarFlag

21.   <UnpackedStructType $GlobalTable $LocalTable $FilestoCheck $BlockInfo
       $Level %TableType $PtrNames %PtrNamesDef $ParentPacking $Packing %TYPE
```

```
   $typevarflag >==>  (21.a)<ArrayType $globaltable $localtable $filestocheck
  $blockinfo $level %tabletype $ptrnames %ptrnamesdef $parentpacking
  $packing $type $typevarflag >    ==> (21.b)<RecordType $globaltable $localtable
   $filestocheck $blockinfo $level %tabletype $ptrnames %ptrnamesdef
  $parentpacking $packing $type> $typevarflag ==> (21.c)<RecordType $globaltype
  $localtable $filestocheck $blockinfo $level %tabletype $ptrnames
  %ptrnamesdef $parentpacking $packing $type $typevarflag >          ==>(21.d) <FileType
  $globaltable $localtable $filestocheck $blockinfo $level %tabletype
  $ptrnames %ptrnamesdef $parentpacking $packing $type> $typevarflag
  ==> (21.e) <SetType $globaltable $localtable $filestocheck $blockinfo $level
  $tabletype $ptrnames $ptrnamesdef $parentpacking $packing %type
  $typevarflag>

2.      <ArrayType $globaltable $localtable $filestocheck $blockinfo $level
        %tabletype $ptrnames %ptrnamesdef $parentpacking $typevarflag>
                ==> 'ARRAY [' <idxtype $globaltable $localtable $filestocheck $blockindo $level
                              %tabletype $ptrnames %ptrnamesdef $packing %arrayactualtype
                              %fileselect $typevarflag >

2.1     <IdxType $globaltable $localtable $filestocheck $blockinfo $level
        %tabletype $ptrnames %ptrnamesdef $packing %array(packing;indextype;
        comptype) %fileselect $typevarflag> ==> <IndexType $globaltable
        $localtable $filestocheck $blockinfo $level %tabletypeindex %indextype>
        <IdxComptype $globaltable $tabletypeindex $filestocheck $blockinfo
        $level %tabletype $ptrnames %ptrnamesdef $packing %comptype %fileselect
        $typevarflag>

2.1.1   <IdxComptype $globaltable %tabletypeindex $filestocheck $blockinfo
        $level %tabletype $ptrnames %ptrnamesdef $packing %comptype %fileselect
        $typevarflag> ==> "]""OF" <ComponentType $globaltable $tabletypeindex
        $filestocheck $blockinfo $level %tabletype $ptrnames %ptrnamesdef
        $packing %comptype %fileselect $typevarflag> | "," <Herecomptype:=
        (actualtype;fileselect packing) <IdxType $globaltable $tabletypeindex
        $filestocheck $blockinfo $level %tabletype $ptrnames %ptrnamesdef
        $packing %actualtype %fileselect $typevarflag>

3.      <IndexType $globaltable $localtable $filestocheck $blockinfo $level
        %tabletypeindex %type.actualtype> ==> <SimpleIdType $globaltable
        $localtable $filestocheck $blockinfo $level %tabletypeindex $unpacked
```

```
%type $1) $typevarflag> <where type.actualtype.typeactTag=ordinal>


<Componentype $globaltable $tabletypeindex $filestocheck $blockinfo
 $level $tabletype $ptrnames %ptrnamesdef $packing %comptype %comptype.
 fileselect $typevarflag> ==> <typeenoter $globaltable $tabletypeindex
 $filestocheck $blockinfo $level %tabletype $ptrnames %ptrnamesdef
 $packing %comptype $typevarflag>


<Recordtype $globaltable $localtable $filestocheck $blockinfo $level
 $tabletype $ptrnames %ptrnamesdef $parentpacking $packing %(record
 (packing;layout;recordtableout;otmptr) fileselect;parentpacking
 $typevarflag> ==> "recend" <FldLst $globaltable $localtable $filestocheck
 $olockinfo $level %tabletype $ptrnames %ptrnamesdef $packing %layout
 %otmptr $1) %recordtableout $typevarflag %fileselect>


<FldLst $globaltable $localtable $filestocheck $blockinfo $level
 %localtable $ptrnames %ptrnamesdef $packing $1) %nil $recordtablein
 %recordtablein $fixpart $taglist  $typevarflag %false> ==> NUL

b   <FldLst $globaltable $localtable $filestocheck $blockinfo $level
 %tabletype $ptrnames %ptrnamesdef $packing %layout %otmptr $recordtablein
 ==> (c)<Compart $globaltable $localtable $filestocheck $blockinfo $level
 %tabletype $ptrnames %ptrnamesdef $packing %layout %otmptr $recordinfo
 $recordtablein %recordtableout $fixpart $taglist %fileselect
 $typevarflag> ==> (d)<Variantpart $globaltable $localttable $filestocheck
 $olockinfo $level %tabletype $ptrnames %ptrnamesdef $packing %layout
 %otmptr $recordinfo $recordtablein %recordtableout $fixpart $taglist
 %fileselect $typevarflag>


a   <Compart $globaltable $localtable $filestocheck $blockinfo $level
 %tabletype $ptrnames %ptrnamesdef $packing $layin %layout %otmptr
 $recordtablein %recordtableout $fixpart $taglist %fileselect
 $typevarflag> ==> <RecordSection $globaltable $localtable $filestocheck
 $blockinfo $level %tabletype $ptrnames %ptrnamesdef $packing $layin
 %layout %otmptr $recirdtablein %recordtableout $fixpart $taglist
 %fileselect $typevarflag> ; <
 <Recordsection $globaltable $localtable $filestocheck $blockinfo
```

```
$level %tabletype1 $ptrnames %ptrnamessection $packing $layin
$layoutsection $otmptr $recordtablein %recordtablesection $fixpart
$taglist %fileselect1> ; <RestrecSection $globaltable $tabletype1
$filestocheck $blockinfo $level %tabletype $ptrnamessection %ptrnamesdef
$packing $layoutsection %layout $otmptr $recordtablesection
%recordtableout $fixpart $taglist %fileselect2 $typevarflag>
        <Here fileselect = fileselect1 or fileselect2 >


<RestrecSection $globaltable $tabletype1 $filestocheck $blockinfo
 $level %tabletype $ptrnamessection %ptrnamesdef $packing $layoutsection
 %layout %otmptr $recordtablesection %recordtableout $fixpart $taglist
 %fileselect2 $typevarflag>
                              ==> <VariantPart $globaltable $tabletype1
 $filestocheck $blockinfo $level %tabletype %ptrnamessection $ptrnames
 $packing $layoutsection %layout %otmptr $recordtablesection
 %recordtableout $fixpart $taglist %fileselect2 $typevarflag> |
 <CompPart $globaltable $tabletype1 $filestocheck $blockinfo $level
   %tabletype $ptrnamessection %ptrnamesdef $packing $layoutsection
 %layout %otmptr $recordtablesection %recordtableout $fixpart $taglist
 %fileselect2 $typevarflag>
.a)     <RestrecSection $globaltable $tabletype1 $filestocheck $blockinfo
 $level %tabletype1 $ptrnamessection %ptrnamessection $packing
 $layoutsection %layoutsection %otmptr $recordtablesection
 %recordtablesection $fixpart $taglist %false $typevarflag>
                                                    ==> NULL

<VariantPart $globaltable $tabletype1 $filestocheck $blockinfo $level
 %tabletype $ptrnames %ptrnamesdef $packing %address( tagtypeactual;
 variantaccesslist) $recordtablein %recordtableout $taglist $layin
 %layout %fileselect $typevarflag>
                                  ==> "CASE" <VariantSelector
 $globaltable/tabletype1 (blockinfo) $filestocheck $level $packing
 $recordtablein %recordtablevariant $layin %layout %discrimination
 %tagtypeactual %taglistelement $fixpart $typevarflag> <VariantList
 $globaltable $tabletype1 $filestocheck $blockinfo $level %tabletype
 $ptrnames %ptrnamesdef $packing %variantaccesslist $recordtablein
 %recordtableout $taglist $taglistelement $tagtypeactual %[]
 %taglistvalue %fileselect $typevarflag> <CheckforCompletnessofCaseLabels
 $tagtypeactual %taglistvalue>


<VariantList $globaltable $tabletype1 $filestocheck $blockinfo $level
```

```
%tabletype $ptrnames $ptrnamesdef $packing $variantaccesslistin %variantaccesslist
          $recordtablein $recordtableout $taglist $taglistelement $tagtypeactual
          $taglistvaluein $taglistvalue $fileselect $typevarflag

                                                    ==> <variant
          $globaltable $tabletype1 $filestocheck $blockinfo $level %tabletype $ptrnames
          $ptrnamesdef $packing $variantaccesslistin %variantaccesslist $recordtablein
          $recordtableout $taglist $taglistelement $tagtypeactual $taglistvaluein
          %taglistvalue %fileselect1  $typevarflag>

o   <VariantList $globaltable $tabletype1 $filestocheck $blockinfo $level%tabletype $ptrnames
          $ptrnamesdef $packing $variantaccesslistin %variantaccesslist $recordtablein
          %recordtableout $taglist $taglistelement $tagtypeactual $taglistvaluein
          %taglistvalue %fileselect1 v fileselect2>

                                                    ==> <Variant $globaltable $tabletype1
          $filestocheck $blockinfo $level %tabletype2 $ptrnames %ptrnames $packing
          $variantaccesslistin %variantaccesslist $recordtablein %recordtable $taglist
          $taglistelement $tagtypeactual $taglistvaluein %taglistvalue1  %fileselect 1
          $level %tabletype $ptrnames,%ptrnamesdef $packing $variantaccesslistin
     %variantaccesslist $recordtable %recordtableout $taglist $taglistelement
          $tagtypeactual $taglistvalue1 %taglistvalue %fileselect2 $typevarflag>

a   <RestVariant $globaltable $tabletype $filestocheck $blockinfo $level %tabletype $ptrnames
          %ptrnames $packing $variantaccesslistin %variantaccesslist $recordtable
          %recordtable $taglist $tagliselement $tagtypeactual $taglistvalue
          %false $typevarflag>
                                                    ==> nil

b   <RestVariant $globaltable $tabletypein $filestocheck $blockinfo $level %tabletypeout
          $ptrnames %ptrnamesdef $packing $variantaccesslistin %variantaccesslist
          $recordtablein %recordtableout $taglist $taglistelement $tagtypeactual
          $taglistvalue1 %taglistvalue2 %fileselect $typevarflag

                                                    ==> <VariantList $globaltable
          $tabletypein $filestocheck $blockinfo $level %tabletypeout $ptrnames %ptrnamesdef
          $packing $variantaccesslistin %variantaccesslist $recordtablein %recordtableout
          $taglist $taglistelement $tagtypeactual $taglistvalue1 %taglistvalue2
                    %fileselect $typevarflag>
    <RecordSection $globaltable $localtable $filestocheck $blockinfo $level %tabletype $ptrnames
          %ptrnamesdef $packing $layin %layout $otmptr $recordtablein %recordtableout
          $fixpart $taglist %type.fileselect $typevarflag>
```

```
                ==> <identlist $recordtablein $namelist > ':'
                    <typeHandler $globaltable $localtable $filestocheck $blockinfo
                $level $tabletype $strnames $strnamesdet $packing $type $car(identlist)
                $typevarflag>
                 <MakeLayoutandRecordtable $layin %layout $namelist $recordtablein
                    %recordtableout $type $fixpart $packing $taglist  $level $tagtypeactual >


.)                      <MakeLayoutandRecordtable $layin %layin $[] $recordtablein
                    %recordtableout $level $typeactual $fixpart $taglist>
                                                              ==> NIL

.a  <MakeLayoutandRecordtable $layin %layout $name ^ namelist $recordtablein %recordtable1 @
                              {$name --> (field(taglist);type;level)} $typeactual $false $taglist
                              $level>
                                  ==> <MakeLayoutandRecordTable $layin %layout $namelist
                              $recordtablein %recordtable1 $typeactual $true $taglist $level>


.c  <MakeLayoutandRecordtable $layin %layin^type $name^namelist $recordtablein %recordtable1
                              @ {$name --> (field(taglist);type;level)} $typeactual $true $taglist $le
                          ==> <MakeLayoutandRecordTable $layin %layin 1 $namelist $recordtablein
                              %recordtable1 $type $true $taglist $level>


1     <VariantSelector $symboltable $filestocheck $level $packing $recordtablein %recordtablein
                       @ {name1 --> (field(taglist);type;level)} $layin %layout %discriminated
                       %type.typeactual %[name1] $fixpart $taglist $typevarflag>
            ==>  Ident %name1 ':' ident %name2 <Here mode=ENV[name2]>
            <Where mode.kind = type and name1 does not belong to filestocheck >
            <Here type=mode.type><Where type.typeactual.ord =char  V  enum  and
            type.typeactual.range=
                    subrange V  all) & <MakeLayout $layin %layout $fixpart $(union(discriminatd;
                    type);(type.typeactual int & subrange>


    <VariantSelector $symboltable $filestocheck $level $packing $recordtablein %recordtablein
                     $layin $layout %freeunion %type.typeactual $[] $fixpart $taglist
                     $typevarflag>
                          ==> "IDENT" $name <HEreMode = mode(enulname)>
                     <WhereMode.kind = type> <HereTypeActual = Mode.type> <Where type.
                     typeactual.typeactualtag = ordinal & type.typeactual.range.rangeselect
```

```
                    = $varname> <MakeLayout $layin $layout $fixpart $(union(free;type)>

.1  <MakeLayout $layin $layin ^false $typeactual>
                                                    ==> NUL

.2  <MakeLayout $layin $layin^$typeactual $true $typeactual>
                                                    ==> NUL

    <Variant $globaltable $localtable $filestocheck $blockinfo $level %tabletype $ptrnames
             %ptrnamesdef $packing $variantaccesslistin %variantaccesslist $recordtablein
              %recordtableout $taglist $taglistelement $tagtypeactual $taglistvaluein
             %taglistvalueout %fileselect $typevarflag>
                                                    ==> <CaseConstantList $globaltable
             /localtable(blockinfo) $tagtypeactual $taglistin %taglistvalueout
             %caseconstlist> <MakeTaglist $taglist $taglistelement $caseconstlist
            %taglistout > ';' '(' <Fldlist $globaltable $localtable $filestocheck $level
                                   %tabletype $ptrnames %ptrnamesdef $packing %layout
                                   $btmptr $recordtablein
             %recordtableout $false $taglistout %fileselect > ")" <MakeVariantAcessList
             $variantaccesslistin %variantaccesslist $caseconstlist $btmptr>

.a  <MakeTaglist $taglist $[] $caseconstlist %taglist>
                                                    ==> NUL

.b  <MakeTaglist $taglist $[name] $caseconstlist %taglist^([name];caseconstlist)
                                                    ==> NUL

    <MakeVariantAccessList $variantaccesslistin %variantaccesslistin1^(caseconst;btmptr)
                           $caseconst^caseconstlist $btmptr>
                                                    ==> <MakeVariantAccessList
                           $variantaccesslistin %variantaccesslistin1 $caseconstlist
                           $btmptr>

.a  <MakeVariantAccessList $variantaccessin %variantaccesslistin $[] $btmptr>
                                                    ==> NUL

    <CaseConstList $symboltable $tagtypeactual $taglistvaluein %taglistvalueout
```

```
<caseconstantlist>
                                ==> <CaseConstant $symboltable $tagtypeactual %constvalue
                <here taglistvalue1 = taglistvalue1~constvalue & caseconstlist =
                [constvalue,> $taglistvalue1 <caseconstantlist1 {,<CaseConstant $symboltable
                $tagtypeactual %constvalue2>} %taglistvalue1 = taglistvalue1~constvalue2
                <caseconstlist1 = caseconstantlist1~constvalue2 <here taglistvalueout =
                taglistvalue1 & caseconstlist = caseconstlist1>

<CaseConstant $symboltable $tagtypeactual %typeactual.range.value1>
                                                                ==> <Constant $symboltable
                %typeactual> <relexedOrdinalTypeEquivalence $tagtypeactual %typeactual
                %ismatch 1 false> <where typeactual.range.value1 DELONGSTO
                tagtypeactual.range>


<SetType $globaltable $localtable $filestocheck $blockinfo $level %tabletype
          $parentpacking $packing %(set(elementtype);false;parentpacking)>
                                                                ==> "SET OF"
           <BaseType $globaltable $localtable $filestocheck $blockinfo $level %tabletype
          $elementype>


<BaseType $globaltable $localtable $filestocheck $blockinfo $level %tabletype $packing
          %type>
                ==> <SimpleIdType $globaltable $localtable $filestocheck $blockinfo $level
          %tabletype $packing %type $1 $false> <where type.typeactual.typeactualtag
         =ordinal >


<filetype $globaltable $localtable $filestocheck $blockinfo $level %tabletype $ptrnames
          %ptrnamesdef $parentpacking $packing %file(filedomain);void);true;parentpacking
          $typevarflag>
                    ==> "FILE OF" <ComponentType $globaltable $localtable $filestocheck
          $blockinfo $level %tabletype $ptrnames %ptrnamesdef $packing %filedomain
          %fileselect1 $typevarflag> <where fileselect1 = false>


a    <PointerType $symboltable $localtable $parentpacking %(pointer(address);false;parentpacking)
               $ptrnames %ptrnamesdef $typevarflag %localtableout
               $level>
                    ==> <DomainType $symboltable $localtable $ptrnames %ptrnamesdef
               %address $typevarflag %localtable>
```

```
1.    <Domaintype $symboltable $localtable $ptrnames %ptrnamesdef %address $typevarflag
                 %localtableout $level>
                                        ==> Ident $name <Here flag:=name BELONGS TO symboltable
                 /localtable(name) block info)> flag?
                                                    true: <address := address(name)> <Where
                                                          address.mode.kind.kindtag = type>
                                                          <localtableout := localtable>
                                                    false: <Typevarflag?
                                                              true: <localtableout :=localtable
                                                                    & {name --> (type;void;level)}>
                                                                    <Ptrnamesdef := ptrnames*name>
                                                                    <Here address := address(name)>
                                                             false: <error>

2.    <TypeId $symboltable %logmode,type>
                                                ==> Ident %name <Here mode=mode(env(name)>
          <Where mode.Kind = type>

3.a   <Var-Dec-Part $globaltable $localtabletype $filestocheck $blockinfo $level %localtabletype
                 %filestocheck>
                                    ==> NUL

3.b   <Var-Dec-Part $globaltable $localtabletype $filestocheck $blockinfo $level %localtablevar
                 %remfilestocheck
                                ==> "VAR" <Here localtable1=localtabletype  and  remfiles1
                =filestocheck> $localtable1 $remfiles1(<Var-Dec $globaltable $localtable
                $remfiles1 $blockinfo $level %localtable2 %remfiles2>;) %localtable1 =
                localtable2 %remfiles1 = remfiles2 <Here localtblevar1=localtable1
                & remfilestocheck = remfiles1><Where remfilestocheck = [ ] >

4.    <Varpec $globaltable $localtable $filestocheck $blockinfo $level %localtablevar %remfilestocnec
                      ==> <identlist $localtable %namelist> ":" <Typedenoter $globaltable $localtabl
             $filestocheck $blockinfo $level %tabletype $ptrnames %ptrnamesdef $unpacked %type>
             <Here mode=(var;type;level)> <putidentlistinlocalenviron $localtable %localtablevar
             $namelist $mode>

4.1.a <Putidentlistinsymboltable $localtable %localtable $[] $mode>
                                                                    ==> NUL
```

```
.1.b  <PutIdentList($globaltable $localtable %localtablel 0 {name --> mode} $name^namelist $mode>
            ==> <PutIdentListInSymbolTable $localtable &localtable $namelist $mode>


.     <Proc-rn-dec-pc $globaltable $localtablevar $globallabels %labelstojump
                      $localtableproc $level $reftablein %reftableout $blockinfo>
                                  ==> <Here localtablein=localtablevar & labelstojump1
                      =labelstojump & reftableinl = reftablein> $localtablein $labelstojump
                      $reftablein1 (<Procdec/rncdec $globaltable $localtablein
                      $globallabels %labelstojumpout &localtableout $level
                      $reftableinl &reftableproc $blockinfo> %localtablein=localtableout
                      %labelstojump1=labelstojump1 U labelstojumpout &reftableinl = reftableproc
                      <Here localtableproc=localtablein & labelstojump=labelstojump1
                      & reftableout = reftableinl>


.     <Proc-dec $globaltable $localtable $globallabels %labelstojump %localtableout $level
                $reftablein %reftableout $blockinfo>
                                  ==> <proc-heading $globaltable $localtable
                $level+1 %proc-table %name $reftablein %reftableinl %localtablehead
                $formaltag $blockinfo %plan %forwardflag>";"<Directive $localtablehead $proctable
                $forwardflag$name $plan %localtableout $blockinfo $level $void>
      <Here localtableout = localtablehead @ {name -->(proc(plan);(void;false);unpacked);level>
                <Proc-block $globaltable/localtableout(blockinfo) $proctable $level+1
                $reftableinl %reftableout $(name;procblock) $globallabels %labelstojump>


.     <Proc-heading $globaltable $localtable $level %proctable %name %plan %forwardflag
                    $reftablein %reftableout>
                                  ==> "PROCEDURE" Ident %name <Forwardentry $localtable $n
                    %presenceflag %mode $forwardtag $blockinfo %localtableout>
                    ?Selection
                      presenceflag : true
                        <Where mode.kind=forwardproc> <Here plan=mode.kind.plan & proctable=
                        mode.kind.environ & reftableout = reftablein & forwardflag = true>
                      presenceflag = false
                        ?SelectionFormaltag
                           false :<Here reftableout=reftablein U {name-->[];[];false;level>
                           true: <reftableout=reftablein & forwardflag = false>
                           <Here plan=[] & proctable = {}> <FormalParlist $globaltable/localtable
                           (blockinfo) $level $formaltag %plan %proctable>
```

```
.1    <FormalPara $symboltable $plan $proctable $level presencetag name
                  $plantableout>
                                    ==> <Identlist
                                          true : <here presencetag = true & node = nodeenv(name)>
                                                 <here localtableout=localtablein-(name==>node,>
                                          false : <here presencetag=false & localtableout =
                                                  localtablein>

.     <FormalParaList $symboltable %plan $proctable $level $formaltag $forvarorflag >
                      ==> '(' <FormalParaSection $symboltable $( ) %plan1 $( ) %proctable1 $level
                              $formaltag
                        $plan1 $proctable1 ';'  <FormalParaSection $symboltable $plan1 $plan2  $proctable1
                        %proctable2 $level $formaltag / %plan2 %proctable2
                        <here plan=plan1 & proctable=proctable1>")"

.     <FormalParaSection $symboltable $planin %planout $proctablein %proctableout $level $formaltag>
                      ==> <Value-Para-Spec $symboltable $planin %planout $proctablein %proctableout $level
                          $formaltag> | <Var-Para-Spec $symboltable $planin %planout $proctablein
                          %proctableout $level $formaltag> | <Proc-Para-Spec $symboltable $planin
                          %planout $proctablein %proctableout $level $formaltag> | <Func-Para-Spec
                          $symboltable $planin %planout $proctablein %proctableout $level $formaltag>

.     <Value-Para-Spec $symboltable $planin %planout $proctablein %proctableout $level
                       $formaltag>
                                  ==> <Identlist $proctablein %namelist> ":" <Typedid $symboltable
                                    %name $type>
                                  ?selectionformaltag
                                    false : <PutIdentListInLocalTable $proctablein %proctableout $namelist
                                                                      $(formalval;type;level)  >
                                            <here proctableout = proctablein >
                                    true : <UpdatePlan $planin %planout $namelist
                                                       $formalpar(simplepar(void;assignment;void;
                                                       (formalval;type;level);0))) >.

0.1.a) <UpdatePlan $planin %planin $( ) $typeactual > ==> lambda

0.1.b) <UpdatePlan $planin %planin1 ^ typeactual $name ^ namelist $typeactual >
                  ==>
```

```
                <IdentPlan $planin $planinl $namelist $typeactual >
))      <VarVarSpec $symboltable $planin $planout $proctablein %proctaableout $level
                    $formaltag >
                              ==> 'VAR' <IdentList $proctablein %namelist > ':'
                    <TypeIdConformantSchema $symboltable $proctablein %proctablel $type $level>
                    <here none = (formalvar;type;level) >
                    formaltag ?
                          false : <PutIdentListInbocaltable $proctablel1 %proctableout
                             $namelist $mode>
                         true  : <PrintePlan $planin $planout $identlist
                             $(((void;typeequivalence;void;mode;0)
                            <here ,proctableout = proctablein >


.1)  <TypeIdConformantSchema $symboltable $proctablein $proctableout %type $level
                             $formaltag >
                                    ==>(a)
                             <TypeId $symboltable %name %type><where type.fileselect =false>
                             <here proctableout = proctablein>
                                          ==>(51.1.b)
                             <ConformantArraySchema $symboltable $proctablein $proctableout
                                                    %type $level $formaltag >


)    <ConformantArraySchema $symboltable $proctablein %proctableout
                            %(typeactual;false;unpacked) $level $formaltag> ==>
                              'ARRAY' '[' <IdxTypeSpec $symboltable $proctablein %proctableout
                                                       %typeactual  >


.1)  <IdxTypeSpec $symboltable $proctablein %proctableout %(contarray(indextype;elementtype))
                  $level $formaltag >
                                    ==>
                       <IndexTypeSpec $symboltable $proctablein %proctable %indextype $level
                                      $formaltag> <IndexType2Spec $symboltable $proctable
                                      %proctableout %elementtype $level $formaltag >


.2)  <IndexType2Spec $symboltable $proctable %proctableout %elementtype $level
                     $formaltag >
                                 ==>(a) 'J' 'UF' <TypeIdConformantSchema $symboltable $proctablein
                                                 %proctableout %elementtype
```

```
                                                        $level $formaltag >

.2b) <IndexTypeSpec $symboltable $proctable $proctableout
                    $actualtype;false;unpacked) $level $formaltag >
                ==> ',' <IndexTypeSpec $symboltable $proctable %proctableout %actualtype
                                       $level $formaltag >

3)   <IndexTypeSpec $symboltable $proctable %proctableout %actualtype $level $formaltag >
                 ==> ident %name1 '..' ident %name2 ':' <OrdinalTypeId $symboltable
                                                        %actualtype>
                  formaltag ?
                          false : <here mode = (boundid;(actualtype;false;void;
                                                  unpacked);level)><here proctableout =
                            proctable @ {name1 ==> mode } @ {name2 ==>mode } >
                          true : <here proctableout = proctablein >

)    <OrdinalTypeId $symboltable %type.actualtype >
                                        ==>< TypeId $symboltable %name %type >
                     <here type.actualtype.actualtagtype = ordinal >

)    <ProcTableSpec $symboltable $plain1 %plain1 ^ simplepar(void;typeequivalence;void;mode;0)
                     $proctablein %proctableout $level $formaltag>
                     ==>
                     <ProcHeading $symboltable $proctablein $level %proctable %name %plan
                                  %forwardflag $1 ) %resttableout true >
                            <here mode = (formalproc(plan);(void;false;unpacked);level)>
                            formaltag ?
                                    true : <here proctableout = proctablein >
                                    false : <here proctableout= proctablein @ {name ==>mode}>

)    <FuncParSpec $symboltable $plain1 $plain1 ^ formalpar(simplepar(void;typeequivalence
                    ;void;mode;0)) $proctablein %proctableout $level  $formaltag > ==>
               <FuncHeading $symboltable $proctablein $level %proctable %name %plan %forwardflag
                            %proctableout true %resulttype >
                 <here mode = (formalfunc(plan);(resulttype;false;unpacked);level) >
               formaltag ?
                            false: <here proctableout = proctablein @ {name  --> mode } >
                            true : <here proctableout = proctablein >
```

```
<Directive $localtable $proctable $name $plan $localtableout $forwardflag $blockinfo
        $level $resulttype >
                                ==> ident $name1 <where name1 = forward>
        <where forwardflag = false >
        blockinfo.blktype :
                    procblk : localtableout = localtable @ {name -->(forwardproc(plan;
                        proctable);(void;false;unpacked);level ) } >
                    funcblk : <here localtableout = localtable @ {name -->
                        (forwardfunc(plan;proctable);(resulttype;false;void;unpacked);
                                                level ) } >

<ProcBlock $globaltable $proctable $level $reftablein &reftableout $blockinfo
                $globallabels &labelstojump >
                                                ==> <Block $globaltable $proctable

                $labelstojump $[ ] $blockinfo $level $reftablein &reftableout >

<FunctionDec $globaltable $localtable $globallabels &labelstojump &localtableout $level
        $reftablein &reftableout $blockinfo >
                                                ==>
        <FuncHeading $globaltable/localtable $level $functable &name &plan
            &forwardflag &localtablehead $formaltag &resulttype $blockinfo > ';'
        (59.a) <Directive $localtablehead $functable $name $plan &localtableout
                $forwardflag $blockinfo $level $ resulttype >
        (59.b) <here localtableout = localtablein @ {name --> (func(plan);
                (resulttype;false;unpacked);level) } >
            <FuncBlock $globaltable/localtable(blockinfo) $functable @ { name -->
                (localfunc(plan);(resulttype;false;unpacked);level+1) } $(name;funcblock)
                $globallabels &labelstojump $reftablein &reftableout >

<FuncHeading $globaltable $localtable $level &functable &name &plan &forwardflag
        $localtableout $formaltag &resulttype $blockinfo>  ==>
    'FUNCTION' ident $name <ForwardEntry $localtable $name &presntflag &mode
                                                &localtableout >
    presentflag :
            TRUE: <where mode.kind = forwardfunc > <here plan = mode.kind &
                functable = mode.kind.environ & forwardflag = true >
            FALSE : <here plan =[ ] & functable ={ }  &forwardflag=false>
```

| <formalparlist $symboltable/localtableout(blockinfo) %plan
%functable $level $formaltag >) ':'
<resulttype $symboltable %resulttypel >

1) <Resulttype $symboltable $type,typeactual >
==>
<typeid $symboltable %name $ type> <where type.typeactual,typeactualtag =
primal : real : pointer >

2) <FunctionBlock $symboltable $functable $level $blockinfo $globallabels %labelstojump
$rettablein %rettableout >
==>
<Block $symboltable $functable $globallabels %labelstojump $( )
$blockinfo $level $rettablein %rettableout>
